NSWI170 – **Computer Systems** – 2023/24 Summer – Martin Svoboda

Assignment **A2: Bead**

Binding Instructions | Well-Meant Advice | Ideas for Thought | Common Mistakes

201. [**Decomposition into functions and classes**] It is expected that the whole code will be decomposed into appropriately designed global functions, classes and their member functions (methods), at different levels of abstraction from the user-friendly control of individual diodes to the actual logic of the running bead itself.

202. [**Class for diode representation**] Individual diodes will be represented as instances of a class that will encapsulate all the necessary data items and expected functionality. The objective is thus nothing else than to design a diode driver that will allow its user-friendly usage, offer a comprehensive interface, and intentionally hide internal technical and implementation details at the same time.

203. [**Private data members**] A good practice when designing classes of our kind is to make all their data members private. Therefore, manipulating with them will not be possible directly from outside, only indirectly through public methods that we design for this purpose. Thanks to that, we will be in control of their content, otherwise they could be changed by anyone and from anywhere without our knowledge. We should also analogously mark as private all potential methods that are purely internal in nature.

204. [**Public diode interface**] The diode class will offer at least two public functions, one to initialize a given diode and the other to change its lighting state. Their task is to wrap the respective low-level Arduino system calls, allowing the users to control the diodes at a higher level of abstraction.

205. [**Logical diode numbers**] In order to really achieve higher abstraction, we will only use logical numbers `0` to `3` for diodes instead of low-level pins everywhere in the code. These numbers, in this order, will correspond to diodes labeled `D1` to `D4`, and therefore pins `led1_pin` to `led4_pin`. The only exception where we can work directly with the pins will be the two functions mentioned above, being the only ones that will internally perform the necessary translation from logical numbers to the corresponding pins.

206. [**Pin number values**] By looking at the attached header file, we can easily find out that pin numbers of individual diodes have values `13` to `10`, but we certainly cannot work with them, regardless of directly or indirectly. We cannot even assume they are potentially ordered or they form a continuous interval.

207. [**Independence on constant values**] We can generalize this idea and extend it to the work with any other constants that we do not fully control. Unless their author has explicitly promised us some special guarantees, we simply cannot make any assumptions about their specific values. Thus we have to be completely independent on them.

208. [**Variable number of diodes**] The entire application must be implemented in a way that we do not enforce or assume any specific fixed number of diodes anywhere. In other words, everything must be universal and work even with a different number of diodes than just `4`. We will derive the specific number of available diodes from the size of the array we created for the purpose of translation of logical diode numbers to their pins.

209. [**Initial state of diodes**] As a part of Arduino initialization, in addition to setting modes of pins, it is also necessary to switch off all the diodes, because we are generally not guaranteed in which current state they may occur.

210. [**Premature system calls**] Although we should actually not get into such a situation with resources we learned, let us emphasize that we must not call any Arduino system functions before the `setup` function, i.e., before its initialization. While such calls would usually pass without any effect on a real Arduino, the emulator in ReCodEx monitors such situations strictly.

211. [**Constructors of diode objects**] The complication described above could specifically occur in a situation where we would like to define explicit constructors for our diode objects, within which we would like to, for example, set modes of pins. Instances of objects in global variables are created at the beginning of the program, i.e., before the `setup` function. If we therefore decide to use such constructors, we need to be aware of that behavior.

212. [**Implicit constructors**] It will actually be even better if we do not use explicit constructors at all. It would only bring us additional technical complications without any advantages, and we did not learn to use them anyway. In other words, we will rely on the implicit constructors that the compiler creates for us automatically. These cause the respective objects are created in such a form that its data members are not initialized in any way. We will therefore arrange setting of their initial values by ourselves via an appropriate initialization method called from the `setup` function.

213. [**Creation of class instances**] If we want to use such a parameterless constructor to create a new diode object, we just need to write `Diode diode`. We do not write the `new` keyword, it has a completely different meaning in the C++ language. Also note that the variable `diode` really contains an instance of the diode itself, not a pointer or some reference to it.

214. [**Diode control**] The diode class will at least offer methods through which we can turn a given diode on or off. However, it would not be a good idea to provide two separate methods in the style of `on` and `off` for this purpose, because then the caller would not be able to dynamically express their request without having to conditionally branch their code somehow. It therefore seems better to offer a method with a `true` / `false` parameter that would allow for such on / off intent distinction easily.

215. [**Diode state preservation**] Let us also add that until we give a given diode a new low-level instruction to light up / light down via the `digitalWrite` call, its state will be preserved unchanged.

216. [**Internal state representation**] If we decide to remember or otherwise express the current internal diode state, it is necessary to represent it at the logical level. Hence, ideally using, e.g., the `bool` data type, not using the low-level constants `LOW` and `HIGH`. Such logical states also need to be correctly transformed into the mentioned `int` constants.

217. [**Named constants**] All constants having some logical meaning in the relation to our application must be declared and appropriately named. Moreover, if it is possible regarding to the other already defined constants, it is also necessary to calculate or otherwise derive their values with the help of such constants, and do that correctly.

218. [**Constancy flag**] Let us also not forget to include the constancy flag as such for each data item that has such a character.

219. [**Global variables**] Considering the programming model offered by Arduino, it is necessary to remember certain information through global variables, although such an approach would not be suitable in the case of standard applications. Otherwise, we would not be able to transfer it between the `setup` function and the individual calls of the `loop` function. However, we will keep the number of such variables to a necessary minimum, we will especially not use global variables in situations where ordinary local ones would in fact suffice.

220. [**Accessing global variables**] We will consider the global variables so significant that we will access them directly and therefore not pass them through function parameters. In the case of more complex objects such as our diodes, this would even not be possible with our knowledge, because their copies would be created and these would become completely separated from the original objects.

221. [**Simple loop content**] However we decompose the logic of our running bead, it is necessary to ensure that the code put directly into the body of the `loop` function is not too large, complicated or technical. Ideally, we should only work here with some basic operations at the highest possible level of abstraction, similarly as was the case with the `main` function.

222. [**Efficient loop implementation**] Function `loop` forms the core of our programming model, therefore it has a key impact on the efficiency of our entire application. Obviously, it is not only about the function itself, but also all the other functions we will directly or indirectly call from within it. It is

performed perpetually over and over, approximately 1000× per second. Depending on the program complexity, this may be an order of magnitude higher as well as lower, though.

223. [**Class for timer representation**] It is expected we solve the functionality for event timing using a suitably designed separate class, too. It will become useful in the future, when we will need to use even more parallel and mutually independent timers at once.

224. [**Timer functionality**] We will also appreciate the possibility to subsequently use different values of the expected intervals, as well as to initialize (start) the timer at any time needed (not only during the `setup` function). Finally, let us add that the timer should only be a passive resource offering us the timing control through an elegant interface, but it should not trigger any actions on its own.

225. [**Time control mechanisms**] In principle, there are two possible mechanisms for timing control. We either remember the time of the previous event and detect if the intended interval has already passed, or we directly schedule the time of the next event and detect if it has already occurred. Although each approach offers certain advantages and disadvantages, the first option will be more suitable, flexible and technically simpler for us.

226. [**Elapsed time detection**] Checking whether a required time interval elapsed needs to be resolved carefully. As we have already discussed, one iteration of the `loop` function can take significantly less time than 1 ms, but it can also take longer. As a result, we are not guaranteed to be able to detect the intended moment of the next event with absolute accuracy, i.e., at the level of individual milliseconds.

227. [**Previous event timestamp**] If we are not able to detect particular time moments precisely, it would be a mistake to save the actual current time of the currently triggered event for the purpose of remembering the previous event timestamp. If we did that, it could cause systematic and repeated delaying that would gradually accumulate and disrupt the required overall regularity of events.

228. [**Current time retrieval**] Calling the `millis` function is not overly time-consuming, but it is also not fast. For this reason, it is advisable to ensure that within one iteration of the `loop` function, we call it only once and we will declare the value thus obtained as the current time valid for the entire given iteration. This will also bring us another benefit. If we were retrieving the current time repeatedly, we could get different values. Depending on our code, this could lead to very uncomfortable error situations caused by such an inconsistency.

229. [**Time value overflow**] For the system time representation, data type `unsigned long` is in particular used. Although it allows us to store time values in milliseconds corresponding up to almost 50 days, sooner or later its overflow will inevitably occur. Therefore, we must be able to treat such situations correctly.

230. [**Sizes of data types**] Just by the way, whenever we want to work with time values, we should consistently use the mentioned type `unsigned long`. Even with the `signed` variant, let alone the `int` type, we would risk problems with a smaller range. Also note that the sizes of these types can vary on different platforms, so while `long` is typically 8 B, it is only 4 B on Arduino.

231. [**Arithmetic over diode numbers**] We should always work with variables in accordance with their data type, but also with the expectations arising from their logical nature. For example, if we represent positions of diodes by their logical numbers, we cannot incorporate them in multiplication nor calculate their absolute value, although such operations would be allowed by integers as such from the technical point of view.

232. [**Prohibition of invalid values**] Analogously, we also cannot work with positions like `-1`, which are not even valid by themselves. We can then generalize this idea so that we can never reserve and use any otherwise normal values to represent erroneous, marginal or perhaps unexpected situations, values, etc. even in other situations.

233. [**Execution of the initial action**] Various initialization actions are expected within the `setup` function, including, e.g., setting the initial position of our bead. Whether we perform the actual lighting up of the corresponding diode immediately, or postpone it with a simple trick and perform it only within the standard running logic of our application in the `loop` function, the first lit diode should be the diode number `0`.

234. [**Timer initialization**] As for setting the initial value of the timer, note that the current time during the `setup` function may no longer necessarily be equal to `0`.

235. [**Zombie actions in loop**] Following our efficiency requirement on the `loop` function, it is always necessary to think very carefully about what actions we want to carry out within it. It would be unacceptable, for example, if we put some conditional action in it while knowing in advance that it would actually be executed only once in the very first iteration and then never again.

236. [**Meaning of the current position**] In order to manage the logic of the running bead, we will probably need to deal with its current position somehow. We should therefore think about what exactly the word *current* actually means in this context. In other words, we should make sure that the current position actually does not reference, for example, the previous position or the following one, on the contrary.

237. [**Template of bead positions**] Although it would actually be a nice idea, we do not want to solve our bead movement problem by creating kind of a template for the sequence of expected bead positions, which we would subsequently iterate through. It does not correspond to the logic of our assignment, which we must of course always respect.

238. [**Logical order of actions**] When realizing the actual bead movement, it is more than appropriate to respect the intuitive order of individual actions, i.e., first turning off the current diode, then moving to the next position, and only then turning on the new diode. Different orderings would be misleading.

239. [**Disallowed system functions**] Our entire application must be programmed in a way that we do not need to use `delay` or `delayMicroseconds` functions, just as we must not block the progress of the `loop` function in any other way. Simply because such attempts would go directly against the very logic of our entire programming model.

240. [**Error signals**] If we encounter an error signal while debugging your program in ReCodEx (which is something else than a return code), the program ran into a serious enough problem that it had to be terminated. The causes can be the following: signal `4` (division by zero or other invalid instruction), `6` (system calls before the `setup` function), `9` (unavailable memory), or `11` (access to unallocated memory, e.g. to items outside of an array).

241. [**Continuity of assignments**] The remaining assignments directly follow each other and we will need to use components such as our diodes, timer or, in the future, e.g., buttons repeatedly. It is therefore suitable to view them as smaller parts of a bigger and gradually built project. From the very beginning, it is worth not underestimating their proposal and, on the contrary, implementing everything in the highest possible quality, so that we do not need to return to them later on. Any further interventions in the form of extensions or even corrections will be much more time-consuming and will only bring the risk of introducing new and difficult-to-debug issues.